Proposal: Binary Large Objects

This proposal was implemented as part of Daffodil 2.5.0

This page is linked from https://s.apache.org/daffodil-blob-feature. If this page content moves, please update that link from https://s.apache.org.

Motivation

DFDL needs an extension that allows data much larger than memory to be manipulated.

A variety of data formats such as for image and video files, consist of fields of what is effectively metadata, surrounding large blocks of data containing compressed image or video data.

An important use case for DFDL is to expose this metadata for easy use, and to provide access to the large data via a streaming mechanism akin to opening a file, rather than including large chunks of a hexBinary string in the infoset, as is common today.

In RDBMS systems, BLOB (Binary Large Object) and CLOB (Character large object) are the types used when the data row returned from an SQL query will not contain the actual value data, but rather a handle that can be used to open/read/write/close the BLOB or CLOB.

DFDL needs at least BLOB capability. This would enable processing of images or video of arbitrary size without the need to every hold all the data in memory.

This also eliminates the limitation on object size.

Requirements

1. Rather than hexBinary data appearing in the infoset, the infoset should include some unique identifier to a location external to the infoset where the hexBinary byte data can be found.
2. The unique identifier must be expressed in DFDL's subset of XML Schema (e.g. things like an attribute containing the unique identifer is not allowed, as attributes aren't part of the DFDL schema language).
3. Accessing, modifying, or creating custom BLOB resources should be possible without the use of Daffodil (e.g. no GUID mappaings stored in Daffodil memory).
4. New API calls or changes to existing API calls should not be required if the BLOB feature is not enabled in a schema (e.g. maintain source backwads compatibility).
5. Suport is only needed for wirting BLOB's out to files, one per BLOB. Future enhancements can be made to support alternative BLOB storage mediums (e.g. databases, offset into original file, API for custom stroage) if necessary.

Proposed Changes

type="xs:anyURI" and dfdl:objectKind

DFDL is extended to allow simple types to have the `xs:anyURI` type. Elements with this type will be treated as BLOB or CLOB objects. The `dfdlx:objectKind` property is added to define what type of object it is. Valid values for this property are "bytes" for binary large objects and "characters" for character large objects.

An example of this usage in a DFDL schema may look something like this:

```
<xs:schema
  xmlns:dfdlx="http://www.ogf.org/dfdl/dfdl-1.0/extensions"
  xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/">

  <xs:element name="data" type="xs:anyURI" dfdlx:objectKind="bytes" dfdl:lengthUnits="bytes" dfdl:length="1024"
/>

</xs:schema>
```

The resulting infoset will look something like this:

```
<data>file:///path/to/blob/data</data>
```

With the 1024 bytes of data being written to a file at location `/path/to/blob/data`.

For this initial proposal, the BLOB URI will always use the `file` scheme and must be absolute. Although this may be a restrictive limitation for some use cases, the flexibility and generality of URI's allows for future enhancement to support different or even custom schemes if needed.

One benefit of this proposal is its simplicity and non-reliance on other DFDL extensions (e.g. one does not need to implement the DFDL layer extension to support this).

Regarding compatibility, any implementations that do not support this extension will likely error with an unsupported `xs:anyURI` type. However, because the other syntax and behavior is very similar to types with `xs:hexBinary,` modifications to switch from xs:anyURI to xs:hexBinary should be minimal.

Daffodil API

With a new simple type defined, some changes to the API are needed to specify where Daffodil should write these new BLOB files. A likely use case is a need to define a different BLOB output directory for each call to `parse()`. Thus, changes to the API must be made to define the output diretory either directly to the `parse()` function or to a parameter already passed to the parse function. Since the `InfosetOutputter` is related parse output, and the BLOB file is a sort of output, it makes the most sense for such definitions that control BLOB file output to added to the `InfosetOutputter`.

Two functions are added to the InfosetOutputter.

The first API function allows a way to set the properties used when creating BLOB files, including the output directory, and prefix/suffix for the BLOB file.

```
/**
 * Set the attributes for how to create blob files.
 *
 * @param dir the Path the the directory to create files. If the directory
 *            does not exist, Daffodil will attempt to create it before
 *            writing a blob.
 * @param prefix the prefix string to be used in generating a blob file name
 * @param suffix the suffix string to be used in generating a blob file name
 */
final def setBlobAttributes(directory: Path, prefix: String, suffix: String)
```

The second API function allows a way for the API user to get a list of all BLOB files that were created during `parse()`.

```
/**
 * Get the list of blob paths that were output in the infoset.
 *
 * This is the same as what would be found by iterating over the infoset.
 */
final def getBlobFiles(): Seq[Path]
```

Note that no changes to the `unparse()` API are required, since the BLOB URI provides all the necessary information to retrieve files containing BLOB data.

Schema Compilation

Schema compilation remains mostly the same. Daffodil will treat elements with the special `xs:anyURI` type similar to a primitive type (e.g. xs:hexBinary) except output will be written to a file in an efficient manner.

Parser Runtime

To support BLOBs, the BLOB parse logic follows these steps:

1. As with hexBinary, determine the starting bitPosition and length of the hexBinary content
2. Create a new BLOB file using directory/prefix/suffix information set in the `InfosetOutputter`.
3. Open the newly created file using using a FileOutputStream. If opening of the file fails, throw a Schema Definition Error.
4. Read length bytes of data from the ParseState dataInputStream and write them out to the FileOutputStream. Chunk the reads into smaller byte lengths to minimize total memory required and to support >2GB of data. If at any point no more bytes are available, throw a PENotEnoughBits parse error. If there is an IOException, throw a Schema Definition Error.
5. Close the file stream.
6. Set the value of the current element to the URI of the File.

Additionally, logic must be created to remove BLOB files if Daffodil backtracks past an already created BLOB. This can be handled by storing the list of BLOB files in the PState, and deleting the appropriate files in the list before resetting back to an early state.

Unparser Runtime

To support BLOBs, the BLOB unparse logic follows these steps:

1. Get the URI from the infoset and the file length. If the length cannot be determined, throw an UnparseError.
2. As with hexBinary, determine the length of the hexBinary content and error if the BLOB file length is larger than the content length

3. Open the File using a FileInputStream. If opening of the file fails, throw an UnparseError
4. Read bytes from the FileInputStream and write them to the UState dataOutputStream. Chunk the reads into smaller byte lengths to minimize total memory required and to support >2GB of data. If at any point there is an IOException, throw an UnparseError.
5. As with hexBinary, write skip bits if the content length is not filled completely.

Note that we are explicitly not removing files after unparsing their data. It is the responsibility of the API user to determine when files are no longer needed and remove them.

DFDL Expression

There are going to be cases where expressions may want to reference elements with type `xs:anyURI`.

This proposal adds the restriction that any expression access to the **data** of a BLOB element is not be allowed. This limitation is really for practical purposes. Presumably, the `xs:anyURI` type is only to be used because the data is very large or meaningless, and so accessing the data is unnecessary. This restriction minimizes complexity since expression do not need to worry about converting blobs to byte arrays or some thing else. If it is later determined that such a feature is needed, this restriction may be lifted. Any access to the data of a BLOB will result in a Schema Definition Error during schema compilation.

This proposal does allow for access to the **length** of a BLOB element. This is almost certainly needed since it is very common in data formats to include both a BLOB payload and the length of that payload. On unparse, we almost certainly need the ability to calculate the length of the BLOB data so that the value can be output in a length field in the data. Fortunately, the content/valueLength functions do not actually query the data, but instead query bitPositions in stored in the infoset. Thus, no changes should be necessary to support this.

Minimal changes my be necessary to make the expression language aware of the xs:anyURI type and act accordingly.

TDML Runner

Because the parsing of BLOBs results in a random URI in the infoset, this provides challenges to the TDML runners ability to compare expected and actual infosets. To resolve this, the TDML Runner will be modified in the following ways:

1. Use the new API to specify a temp directory for BLOBs to be stored

2. Perform type aware comparisons for the `xs:anyURI` type, similar to what we do now for `xs:date`, xs:dateTime, and `xs:time`. Type awareness will be enable by using the `xsi:type` attribute on the expected infoset, since Daffodil does not currently supprt adding `xsi:type` information to the actual infoset as of yet. And example looks something like:

   ```
   <tdml:dfdlInfoset>
     <data xsi:type="xs:anyURI">path/to/blob/data</data>
   </tdml:dfdlInfoset>
   ```

   During type aware comparisons, the TDML Runner will extract and modify the path (e.g. find the file and convert it to absolute in the infoset) to be suitable for use in logic similar to finding files using the `type="file"` attribute for expected infosets. Once the expected file is found, it will compare the contents of that file with the contents of the URI specified in the actual infoset and report any differences as usual.

3. After a test completes, delete all BLOB files listed in the InfosetOutputter

Command Line Interface

The CLI will be modified to use the new BLOB API on the `InfosetInputter` to set the BLOB directory to "daffodil-blobs" appropriately. A future enhancement may add an option to configure a different blob directory.

The CLI will never delete blob files.